Тема

«Нововведения в java 8»

Состав

Лямбда выражения

StreamAPI

Нововведения в датах

Лямбда в java8
Методы интерфейсов по умолчанию

Java 8 позволяет вам добавлять неабстрактные реализации методов в интерфейс, используя ключевое слово `default`. Эта фича также известна, как методы расширения. Вот наш первый пример:

```
interface Formula {
    double calculate(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

Кроме абстрактного метода `calculate` интерфейс `Formula` также определяет метод по умолчанию `sqrt`. Классы, имплементирующие этот интерфейс, должны переопределить только абстрактный метод `calculate`. Метод по умолчанию `sqrt` будет доступен без переопределения.

```
Formula formula = new Formula() {
    @Override
```

```
        public double calculate(int a) {
            return sqrt(a * 100);
        }
    };

    formula.calculate(100);     // 100.0
    formula.sqrt(16);           // 4.0
```

`formula` реализован как анонимный объект. Этот код довольно избыточен: целых 6 строчек для такого простого вычисления как `sqrt(a * 100)`. В следующем разделе мы увидим, что в Java 8 есть куда более изящный способ реализации объектов с одним методом.

Лямбда-выражения

Давайте начнем с простого примера: сортировка массива строк в предыдущих версиях языка.

```
List<String> names = Arrays.asList("peter", "anna", "mike",
"xenia");

Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
```

```
});
```

Статический метод `Collections.sort` принимает список и компаратор, который используется для сортировки списка. Наверняка вам часто приходилось создавать анонимные компараторы для того чтобы передать их в метод.

Java 8 предоставляет гораздо более короткий синтаксис — лямбда-выражения, чтобы вам не приходилось тратить время на создание анонимных объектов:

```
Collections.sort(names, (String a, String b) -> {
        return b.compareTo(a);
});
```

Как видите, код гораздо короче и куда более читаем. И его можно сделать еще короче:

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

Для однострочных методов вы можете опустить скобки {} и ключевое слово `return`. Так еще короче:

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Компилятору известны типы параметров, поэтому их можно тоже опустить. Давайте посмотрим, как еще могут использовать лямбда-выражения.

Функциональные интерфейсы

Как лямбда-выражения соответствуют системе типов языка Java? Каждой лямбде соответствует тип, представленный интерфейсом. Так называемый *функциональный интерфейс* должен содержать **ровно один абстрактный метод**. Каждое лямбда-выражение этого типа будет сопоставлено объявленному методу. Также, поскольку методы по умолчанию не являются абстрактными, вы можете добавлять в функциональный интерфейс сколько угодно таких методов.

Мы можем использовать какие угодно интерфейсы для лямбда-выражений, содержащие ровно один абстрактный метод. Для того, чтобы гарантировать, что ваш интерфейс отвечает этому требованию, используется аннотация `@FunctionalInterface`.

Компилятор осведомлен об этой аннотации, и выдаст ошибку компиляции, если вы добавите второй абстрактный метод в функциональный интерфейс.

Пример:

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}
```

```
Converter<String, Integer> converter = (from) ->
Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted);    // 123
```

Имейте в виду, что этот код останется корректным даже если убрать аннотацию `@FunctionalInterface`.

Ссылки на методы и конструкторы

Предыдущий пример можно упростить, если использовать статические ссылки на методы:

```
Converter<String, Integer> converter = Integer::valueOf;
Integer converted = converter.convert("123");
System.out.println(converted);    // 123
```

Java 8 позволяет вам передавать ссылки на методы или конструкторы. Для этого нужно

использовать ключевое слово `::`. Предыдущий пример иллюстрирует передачу ссылки на статический метод. Однако мы также можем ссылаться на экземплярный метод:

```
class Something {
    String startsWith(String s) {
        return String.valueOf(s.charAt(0));
    }
}
```

```
Something something = new Something();
Converter<String, String> converter = something::startsWith;
String converted = converter.convert("Java");
System.out.println(converted);    // "J"
```

Давайте посмотрим, как передавать ссылки на конструкторы. Сперва определим бин с несколькими конструкторами:

```
class Person {
    String firstName;
    String lastName;

    Person() {}
```

```
    Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

Затем определим интерфейс фабрики, которая будет использоваться для создания новых персон:

```
interface PersonFactory<P extends Person> {
    P create(String firstName, String lastName);
}
```

Теперь вместо реализации интерфейса мы соединяем все вместе при помощи ссылки на конструктор:

```
PersonFactory<Person> personFactory = Person::new;
Person person = personFactory.create("Peter", "Parker");
```

Мы создаем ссылку на конструктор с помощью `Person::new`. Компилятор автоматически выбирает подходящий конструктор, сигнатура которого совпадает с сигнатурой `PersonFactory.create`.

Области действия лямбд

Доступ к переменным внешней области действия из лямбда-выражения очень схож к доступу из анонимных объектов. Вы можете ссылаться на переменные, объявленные как `final`, на экземплярные поля класса и статические переменные.

```
final    int num = 1;
Converter<Integer, String> stringConverter = (from) ->
String.valueOf(from + num);

stringConverter.convert(    2);     // 3
```

Но в отличии от анонимных объектов, переменная `num` не обязательно должна быть объявлена как `final`. Такой код тоже сработает:

```
int    num = 1;
Converter<Integer, String> stringConverter = (from) ->
String.valueOf(from + num);

stringConverter.convert(    2);     // 3
```

Однако переменная `num` должна все равно оставаться неизменяемой. Следующий код **не**скомпилируется:

```
int    num = 1;
Converter<Integer, String> stringConverter = (from) ->
String.valueOf(from + num);
num =    3;
```

Запись в переменную `num` в пределах лямбда-выражения также запрещена.

Доступ к полям и статическим переменным

В отличии от локальных переменных, мы можем записывать значения в экземплярные поля класса и статические переменные внутри лямбда-выражений. Это поведение хорошо знакомо по анонимным объектам.

```
class Lambda4 {
    static int outerStaticNum;
    int outerNum;

    void testScopes() {
        Converter<Integer, String> stringConverter1 = (from) -> {
            outerNum = 23;
            return String.valueOf(from);
        };

        Converter<Integer, String> stringConverter2 = (from) -> {
            outerStaticNum = 72;
            return String.valueOf(from);
        };
    }
}
```

Доступ к методам интерфейсов по умолчанию

Помните пример с формулой из первого раздела? В интерфейсе `Formula` определен метод по умолчанию `sqrt`, который доступен из каждой имплементации интерфейса, включая анонимные объекты. Однако это не сработает в лямбда-выражениях.

Внутри лямбда-выражений запрещено обращаться к методам по умолчанию. Следующий код не скомпилируется:

```
Formula formula = (a) -> sqrt( a *     100);
```

StreamAPI
Встроенные функциональные интерфейсы

JDK 1.8 содержит множество встроенных функциональных интерфейсов. Некоторые из них хорошо известны по предыдущим версиям языка,
например, `Comparator` или `Runnable`. Все эти интерфейсы были поддержаны в лямбдах добавлением аннотации `@FunctionalInterface`.

Однако в Java 8 также появилось много новых функциональных интерфейсов, которые призваны облегчить вам жизнь. Некоторые интерфейсы хорошо известны по библиотеке Google Guava. Даже если вы незнакомы с этой библиотекой, вам стоить взглянуть, как эти интерфейсы были дополнены некоторыми полезными методами расширений.

Предикаты

Предикаты — это функции, принимающие один аргумент, и возвращающие значение типа boolean. Интерфейс содержит различные методы по умолчанию, позволяющие строить сложные условия (`and`, `or`, `negate`).

```
Predicate<String> predicate = (s) -> s.length() >     0;
```

```
predicate.test(     "foo");              // true
```

```
predicate.negate().test(     "foo");     // false
```

```
Predicate<Boolean> nonNull = Objects::nonNull;
```

```
Predicate<Boolean> isNull = Objects::isNull;
```

```
Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNotEmpty = isEmpty.negate();
```

Функции

Функции принимают один аргумент и возвращают некоторый результат. Методы по умолчанию могут использоваться для построения цепочек вызовов (`compose`, `andThen`).

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString =
toInteger.andThen(String::valueOf);

backToString.apply(  "123");     // "123"
```

Поставщики

Поставщики (suppliers) предоставляют результат заданного типа. В отличии от функций, поставщики не принимают аргументов.

```
Supplier<Person> personSupplier = Person::  new;
personSupplier.get();      // new Person
```

Потребители

Потребители (consumers) представляют собой операции, которые производятся на одним входным аргументом.

```
Consumer<Person> greeter = (p) -> System.out.println(  "Hello, "
+ p.firstName);
greeter.accept(   new Person("Luke", "Skywalker"));
```

Компараторы

Компараторы хорошо известны по предыдущим версиям Java. Java 8 добавляет в интерфейс различные методы по умолчанию.

```
Comparator<Person> comparator = (p1, p2) ->
p1.firstName.compareTo(p2.firstName);
```

```
Person p1 =    new Person("John", "Doe");
Person p2 =    new Person("Alice", "Wonderland");
```

```
comparator.compare(p1, p2);                 // > 0
comparator.reversed().compare(p1, p2);      // < 0
```

Опциональные значения

Опциональные значения (optionals) не являются функциональными интерфейсами, однако являются удобным средством предотвращения NullPointerException. Это важная концепция, которая понадобится нам в следующем разделе, поэтому давайте взглянем, как работают опциональные значения.

Опциональные значение — это по сути контейнер для значения, которое может быть равно null. Например, вам нужен метод, который возвращает какое-то значение, но иногда он должен возвращать пустое значение. Вместо того, чтобы возвращать null, в Java 8 вы можете вернуть опциональное значение.

```
Optional<String> optional = Optional.of(    "bam");
```

```
optional.isPresent();                 // true
optional.get();                       // "bam"
optional.orElse(    "fallback");      // "bam"
```

```
optional.ifPresent((s) -> System.out.println(s.charAt(0)));
// "b"
```

Потоки

Тип `java.util.Stream` представляет собой последовательность элементов, над которой можно производить различные операции. Операции над потоками бывают или промежуточными (intermediate) или конечными (terminal). Конечные операции возвращают результат определенного типа, а промежуточные операции возвращают тот же поток. Таким образом вы можете строить цепочки из несколько операций над одним и тем же потоком. Поток создаются на основе источников, например типов, реализующих `java.util.Collection`, такие как списки или множества (ассоциативные массивы не поддерживаются). Операции над потоками могут выполняться как последовательно, так и параллельно.

Сначала давайте посмотрим, как работать с потоком последовательно. Сперва создадим источник в виде списка строк:

```
List<String> stringCollection = new ArrayList<>();
stringCollection.add("ddd2");
stringCollection.add("aaa2");
stringCollection.add("bbb1");
stringCollection.add("aaa1");
stringCollection.add("bbb3");
stringCollection.add("ccc");
stringCollection.add("bbb2");
stringCollection.add("ddd1");
```

В Java 8 вы можете быстро создавать потоки, используя вызовы `Collection.stream()` или `Collection.parallelStream()`. Следующие разделы объясняют наиболее распространенные операции над потоками.

Filter

Операция Filter принимает предикат, который фильтрует все элементы потока. Эта операция является *промежуточной*, т.е. позволяет нам вызвать другую операцию (например, `forEach`) над результатом. ForEach принимает функцию, которая вызывается для каждого элемента в (уже отфильтрованном) поток. ForEach является *конечной* операцией. Она не возвращает никакого значения, поэтому дальнейший вызов потоковых операций невозможен.

```
stringCollection
    .stream()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa2", "aaa1"
```

Sorted

Операция Sorted является *промежуточной* операцией, которая возвращает отсортированное представление потока. Элементы сортируются в обычном порядке, если вы не предоставили свой компаратор:

```
stringCollection
    .stream()
    .sorted()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);
```

```
// "aaa1", "aaa2"
```

Помните, что `sorted` создает всего лишь отсортированное представление и не влияет на порядок элементов в исходной коллекции. Порядок строк в `stringCollection` остается нетронутым:

```
System.out.println(stringCollection);
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1
```

Map

Промежуточная операция `map` преобразовывает каждый элемент в другой объект при помощи переданной функции. Следующий пример преобразовывает каждую строку в строку в верхнем регистре. Однако вы так же можете использовать `map` для преобразования каждого объекта в объект другого типа. Тип результирующего потока зависит от типа функции, которую вы передаете при вызове `map`.

```
stringCollection
    .stream()
    .map(String::toUpperCase)
    .sorted((a, b) -> b.compareTo(a))
    .forEach(System.out::println);
// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

Match

Для проверки, удовлетворяет ли поток заданному предикату, используются различные

операции сопоставления (match). Все операции сопоставления являются *конечными* и возвращают результат типа boolean.

```
boolean anyStartsWithA =
    stringCollection
        .stream()
        .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA);      // true
```

```
boolean allStartsWithA =
    stringCollection
        .stream()
        .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA);      // false
```

```
boolean noneStartsWithZ =
    stringCollection
        .stream()
        .noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ);      // true
```

Count

Операция `Count` является *конечной* операцией и возвращает количество элементов в потоке. Типом возвращаемого значения является `long`.

```
long startsWithB =
    stringCollection
        .stream()
        .filter((s) -> s.startsWith("b"))
        .count();

System.out.println(startsWithB);    // 3
```

Reduce

Эта конечная операция производит свертку элементов потока по заданной функции. Результатом является опциональное значение.

```
Optional<String> reduced =
    stringCollection
        .stream()
        .sorted()
        .reduce((s1, s2) -> s1 + "#" + s2);
```

```
reduced.ifPresent(System.out::println);
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"
```

Параллельные потоки

Как уже упоминалось выше, потоки могут быть последовательными и параллельными. Операции над последовательными потоками выполняются в одном потоке процессора, над параллельными — используя несколько потоков процессора.

Следующие пример демонстрирует, как можно легко увеличить скорость работы, используя параллельные потоки.

Сперва создадим большой список из уникальных элементов:

```
int max = 1000000;
List<String> values = new ArrayList<>(max);
for (int i = 0; i < max; i++) {
    UUID uuid = UUID.randomUUID();
    values.add(uuid.toString());
}
```

Теперь измерим время сортировки этого списка.

Последовательная сортировка

```
long t0 = System.nanoTime();
```

```
long count = values.stream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("sequential sort took: %d ms",
millis));

// sequential sort took: 899 ms
```

Параллельная сортировка

```
long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("parallel sort took: %d ms",
millis));
```

```
// parallel sort took: 472 ms
```

Как вы можете видеть, оба куска кода практически идентичны, однако параллельная сортировка почти в два раза быстрее. Все, что вам нужно сделать, это заменить вызов `stream()` на `parallelStream()`.

Ассоциативные массивы

Как уже упоминалось, ассоциативные массивы (maps) не поддерживают потоки. Вместо этого ассоциативные массивы теперь поддерживают различные полезные методы, которые решают часто встречаемые задачи.

```
Map<Integer, String> map = new HashMap<>();
for (int i = 0; i < 10; i++) {
    map.putIfAbsent(i, "val" + i);
}
map.forEach((id, val) -> System.out.println(val));
```

Этот код в особых комментариях не нуждается: `putIfAbsent` позволяет нам не писать дополнительные проверки на null; `forEach` принимает потребителя, который производит операцию над каждым элементом массива.

Этот код показывает как использовать для вычислений код при помощи различных функций:

```
map.computeIfPresent(3, (num, val) -> val + num);
map.get(3);             // val33
map.computeIfPresent(9, (num, val) -> null);
```

```
map.containsKey(9);     // false
```

```
map.computeIfAbsent(23, num -> "val" + num);
map.containsKey(23);    // true
```

```
map.computeIfAbsent(3, num -> "bam");
map.get(3);             // val33
```

Затем мы узнаем, как удалить объект по ключу, только если этот объект ассоциирован с ключом:

```
map.remove(3, "val3");
map.get(3);             // val33
```

```
map.remove(3, "val33");
map.get(3);             // null
```

Еще один полезный метод:

```
map.getOrDefault(42, "not found");  // not found
```

Объединить записи двух массивов? Легко:

```
map.merge(9, "val9", (value, newValue) ->
value.concat(newValue));
```

```
map.get(9);             // val9
```

```
map.merge(9, "concat", (value, newValue) ->
value.concat(newValue));
```

```
map.get(9);             // val9concat
```

В случае отсутствия ключа `Merge` создает новую пару ключ-значение. В противном случае — вызывает функцию объединения для существующего значения.

API для работы с датами

Java 8 содержит совершенно новый API для работы с датами и временем, расположенный в пакете `java.time`. Новый API сравним с библиотекой Joda-Time, однако имеются отличия. Следующие разделы рассказывают о наиболее важных частях нового API.

Clock

Тип `Clock` предоставляет доступ к текущей дате и времени. Этот тип знает о часовых поясах и может использоваться вместо вызова `System.currentTimeMillis()` для возвращения миллисекунд. Такая точная дата также может быть представлена классом `Instant`. Объекты этого класса могут быть использованы для создания объектов устаревшего типа
`java.util.Date`.

```
Clock clock = Clock.systemDefaultZone();
```

```
long   millis = clock.millis();
```

```
Instant instant = clock.instant();
```

```
Date legacyDate = Date.from(instant);   // legacy java.util.Date
```

Часовые пояса

Часовые пояса представлены типом `ZoneId`. Доступ к ним можно получить при помощи

статических фабричных методов. Часовые пояса содержат смещения, которые важны для конвертации дат и времени в местные.

```
System.out.println(ZoneId.getAvailableZoneIds());
// prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");
System.out.println(zone1.getRules());
System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]
// ZoneRules[currentStandardOffset=-03:00]
```

LocalTime

Тип `LocalTime` представляет собой время с учетом часового пояса, например, 10pm или 17:30:15. В следующем примере создаются два местных времени для часовых поясов, определенных выше. Затем оба времени сравниваются, и вычисляется разница между ними в часах и минутах.

```
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2));  // false
```

```
long hoursBetween = ChronoUnit.HOURS.between(now1, now2);
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);
```

```
System.out.println(hoursBetween);   // -3
System.out.println(minutesBetween); // -239
```

Тип LocalTime содержит различные фабричные методы, которые упрощают создание новых экземпляров, а также парсинг строк.

```
LocalTime late = LocalTime.of(23, 59, 59);
System.out.println(late);       // 23:59:59

DateTimeFormatter germanFormatter =
    DateTimeFormatter
        .ofLocalizedTime(FormatStyle.SHORT)
        .withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);
System.out.println(leetTime);   // 13:37
```

LocalDate

Тип `LocalDate` представляет конкретную дату, например, 2014-03-11.
Объекты `LocalDate` неизменяемы и являются аналогом `LocalTime`. Пример демонстрирует вычисление новой даты путем сложения или вычитания дней, месяцев или годов. Помните, что каждая операция возвращает новый экземпляр.

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
LocalDate yesterday = tomorrow.minusDays(2);

LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
System.out.println(dayOfWeek);    // FRIDAY
```

Создание экземпляра `LocalDate` путем парсинга строки:

```
DateTimeFormatter germanFormatter =
    DateTimeFormatter
        .ofLocalizedDate(FormatStyle.MEDIUM)
        .withLocale(Locale.GERMAN);

LocalDate xmas = LocalDate.parse("24.12.2014",
germanFormatter);
System.out.println(xmas);    // 2014-12-24
```

LocalDateTime

Тип `LocalDateTime` представляет собой комбинацию даты и времени.

Объекты `LocalDateTime`неизменяемы и работают аналогично `LocalTime` и `LocalDate`. Мы можем использовать различные методы для извлечения конкретных значений из даты-времени:

```
LocalDateTime sylvester = LocalDateTime.of(2014,
Month.DECEMBER, 31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
System.out.println(dayOfWeek);      // WEDNESDAY

Month month = sylvester.getMonth();
System.out.println(month);          // DECEMBER

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE_OF_DAY);
System.out.println(minuteOfDay);    // 1439
```

Путем добавления информации о часовом поясе мы можем получить `Instant`.

```
Instant instant = sylvester
        .atZone(ZoneId.systemDefault())
        .toInstant();

Date legacyDate = Date.from(instant);
```

```
System.out.println(legacyDate);  // Wed Dec 31 23:59:59 CET 2014
```

Форматирование даты-времени работает так же, как и форматирование даты или времени. Мы можем использовать библиотечные или свои собственные шаблоны.

```
DateTimeFormatter formatter =
        DateTimeFormatter
                .ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);
String string = formatter.format(parsed);
System.out.println(string);  // Nov 03, 2014 - 07:13
```

В отличии от `java.text.NumberFormat`, новый `DateTimeFormatter` является неизменяемым и потокобезопасным.

Подробно о синтаксисе шаблонов можно почитать здесь.

Аннотации

Аннотации в Java 8 являются повторяемыми. Давайте сразу посмотрим пример, чтобы понять, что это такое.

Сперва мы определим аннотацию-обертку, которая содержит массив аннотаций:

```
@interface Hints {
    Hint[] value();
```

```
}
```

```
@Repeatable (Hints.class)
@interface Hint {
    String value();
}
```

Java 8 позволяет нам использовать множество аннотаций одного типа путем указания аннотации `@Repeatable`.

Вариант 1: использовать аннотацию-контейнер (старый способ)

```
@Hints ({@Hint("hint1"), @Hint("hint2")})
class Person {}
```

Вариант 2: использовать повторяемую аннотацию (новый способ)

```
@Hint ("hint1")
@Hint ("hint2")
class Person {}
```

При использовании варианта 2 компилятор автоматически подставляет аннотацию `@Hints`. Это важно при чтении информации об аннотациях через рефлексию.

```
Hint hint = Person.class.getAnnotation(Hint.class);
System.out.println(hint);   // null
```

```
Hints hints1 = Person.class.getAnnotation(Hints.class);
System.out.println(hints1.value().length);    // 2

Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class);
System.out.println(hints2.length);    // 2
```

Хотя мы никогда не объявляли аннотацию @Hints в классе `Person`, она доступна нам при вызове `getAnnotation(Hints.class)`. Однако более удобным является метод `getAnnotationsByType`, который напрямую предоставляет доступ ко всем аннотациям `@Hint`.

Более того, аннотации в Java 8 можно использовать еще на двух элементах:

```
@Target ({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@interface MyAnnotation {}
```

Материалы для подготовки

1. https://www.ibm.com/developerworks/ru/library/j-java8lambdas/
2. https://habrahabr.ru/company/luxoft/blog/270383/

Вопросы для самоконтроля

1) В чем смысл введения Stream API?
2) Какие преимущества и ограничения дают лямбда выражения?
3) Перечислите известные функциональные интерфейсы?